Abduction in Machine Learning

[Tarski, 1936] A. Tarski. Über den Begriff der logischen Folgering, *Actes du Congrès Int. de Philosophie Scientifique* 7:1–11, 1936. Translated into English as On the concept of logical consequence. In *Logic, Semantics, Metamathematics*, A. Tarski, pp. 409–420, Clarendon Press, Oxford, 1956.
[Zadrozny, 1991] W. Zadrozny. On rules of abduction. IBM Research Report, August 1991.

F. BERGADANO, V. CUTELLO AND D. GUNETTI

ABDUCTION IN MACHINE LEARNING

1 INTRODUCTION

Both inductive learning and abductive reasoning start from specific facts or observations and produce some explanation of these facts. Both may be described as forms of defeasible reasoning from effects to causes. There are some differences, but they are minor and due to different understandings of the notions of observation and explanation (see for instance [Bergadano and Besnark, 1994]). We build on the general notions developed in the introductory Chapter, taking what was labeled there as the *syllogistic* view, in the sense that we isolate the differences between abduction and induction based on syntactic considerations.

Briefly, induction sees examples as *instances of a concept* and explanations as general *concept descriptions*, whereas abduction sees examples as specific observations and explanations as other specific facts that are true, and cause the observations to occur. As a typical case for induction, an example could be the description of a specific bird, and a concept description could be a rule such as

$$\text{bird}(X) \leftarrow \text{small}(X) \wedge \text{part\_of}(X, Y) \wedge \text{wing}(Y)$$

with the obvious intended meaning. For abduction, an observation could be the fact that some particular bird does not fly, and an explanation could be the fact that that particular bird has a broken wing.

In both cases, the relevant logical relation is that the explanation, together with the domain knowledge, implies the observation. This aspect is developed in depth in the introductory Chapter.

The Machine Learning literature has largely ignored such similarities, or has produced studies that emphasize the differences that are present [Console and Saitta, 1993; Flach, 1992]. On the other hand, abduction has been used as an effective technique within the underlying inductive framework of Machine Learning methods. Two different approaches, which we think are quite representative of the Machine Learning view of abductive reasoning, can be identified.

- First, abduction has been used to guide search in top-down relational learning. The idea has evolved from explanation-based learning methods, restricting the inductive hypotheses to be logical consequences of a given domain theory. This is then generalized by allowing inductive hypotheses to be obtained from the domain theory by either deductive or abductive reasoning.

- Second, abduction has been used to generate missing examples in relational learning. In fact, it is intrinsic to the nature of induction that the input data is incomplete: some examples are given, but not all. If some particular

D. M. Gabbay and R. Kruse (eds.), Abductive Reasoning and Learning, 197–229.

examples are missing, existing relational learning methods may however encounter serious problems, and abductive generation of these missing examples may be an effective solution.

Top-down relational learning algorithms suffer from the difficulty of searching a space of possible inductive hypotheses that is usually very large. In the first relational learning systems, and also in more recent approaches such as Foil [Quinlan, 1990], the problem was addressed with heuristics of a statistical nature. Top-down systems start from very general concept descriptions, and then try to obtain consistent rules with a number of specialization steps. Typically, heuristics would favor specializations that exclude negative examples while still covering a large number of positive examples. Such heuristics may be misleading, and may also be insufficient for an adequate pruning of the hypothesis space. The Explanation-Based Learning (EBL) paradigm restricts the concept descriptions that may be possibly produced to the logical consequences of a so-called *domain theory*, that is given as input to the learning system [Mitchell *et al.*, 1986; Bergadano and Giordana, 1988; Pazzani and Kibler, 1992]. Abduction has been used in this framework as follows: given the domain theory, the concept descriptions that may be obtained via abductive reasoning are considered as possibly true, and should be evaluated inductively on the basis of the available examples. Studies that follow this scheme may be found in [Bergadano *et al.*, 1989; Cohen, 1992; Cohen, 1994].

A well known problem in Machine Learning is to provide a learning method with an "adequate" set of examples of the target concept. Here "adequate" informally means that the training set should contain all those examples required to successfully complete the learning task, and no more. Obviously, this is a very hard condition to achieve, since usually it is not possible to know in advance exactly which examples are (and which are not) significant for learning a concept. As a consequence, the learning task may turn out to be too slow (if too many examples are given) and/or may fail (if the examples are not significant). This problem is particularly serious for an important class of learning methods: *Relational Learning Algorithms* based on an extensional interpretation of sub-predicates and recursion [Quinlan, 1990; Bergadano and Giordana, 1988; Pazzani and Kibler, 1992]. In these methods, the learning procedure can not only fail or be too slow, but also produce wrong results: the description of the target concept synthesized by the system may entail some of the negative examples given to the system.

It is our objective to show how abduction can be used to fix the above problem. An abductive procedure is used to query the user for any example that may be missing, depending on the hypothesis space that has been defined and the given examples. A similar technique has been used before, for example in [DeRaedt and Bruynooghe, 1991; Shapiro, 1983], to query the user for missing values allowing a single example to be covered. In our case, abduction will be systematically applied over the whole hypothesis space. As a result, the learning systems turns out to be

correct and sufficient, in the sense that the learned description does not entail any of the negative examples and such a description can always be found if it exists. We also show how this technique can be adapted to the problem of program testing: a combination of induction and abduction can be used to generate an "adequate" test set for a program under testing.

2 ABDUCTION AND INDUCTION

Abductive and Inductive Reasoning in Artificial Intelligence are considered to be distinct and have generated separate fields of study. After a simple analysis, one finds in effect distinct inference schemes.

- For induction: $\dfrac{P(a)}{\forall x P(x)}$
- For abduction: $\dfrac{P(x) \rightarrow Q(x) \qquad Q(a)}{P(a)}$

As stated in the introductory Chapter, a deeper analysis, however, suggests that the difference between the two schemes is not always easy to state. For instance, using the tautology

$$\Phi \equiv (\forall x P(x)) \rightarrow P(a)$$

one gets:

$$\frac{\Phi \qquad P(a)}{\forall x P(x)}$$

as an abductive inference step, but it actually has the same premises and conclusions of the inductive inference scheme.

It would then seem that the abductive scheme includes simple forms of inductive reasoning. In [Bergadano and Besnark, 1994] the authors start from the above considerations and then isolate some minor differences of abductive and inductive reasoning within the same framework, that is determined by the above inference rules. In the same paper, a formalization based on non-monotonic logic is developed. Here, we simply note that induction and abduction are indeed very similar inference schemes. A deeper analysis is found in the introductory Chapter of the present Book, where general similarities are noted, and syntactic, inferential and semantic differences are considered.

However, the Machine Learning literature has not used abduction and induction as synonyms. The main keyword in learning is induction, and abductive reasoning is rather used as an additional technique for solving particular problems. We discuss in the following sections two of such uses of abduction in Machine Learning. In particular in the next section we survey its use as a form of bias for guiding the search in a top-down specialization. Then, we show how abductive reasoning and queries can solve some relevant problems in relational top-down learning, of both binary and fuzzy predicates.

3 GUIDING SEARCH WITH ABDUCTION

One use of abduction in Machine Learning is related to the problem of guiding search in a top-down specialization. As the number of concept descriptions that may possibly be generated is, in general, very large, and since even the number of descriptions that are consistent with the examples can be large, learning systems need extra-evidential criteria to prune the search space. Deductive inference with a domain theory has been used to this purpose in ML-SMART [Bergadano and Giordana, 1988] and FOCL [Pazzani and Kibler, 1992]. Similarly, a domain theory could be used abductively within the same framework as in [Bergadano *et al.*, 1989] and in [Cohen, 1992; Cohen, 1994].

Before we can suitably describe the details of abductive inference to prune the space of possible concept descriptions, we need to define a type of analytic learning called Explanation-based Learning (EBL), which has received much attention during the late eighties. A survey of EBL is found in [Ellman, 1989].

EBL needs as input one or more positive examples, as well as a so-called *domain theory*, which includes relevant prior knowledge. A resolution proof of the positive examples is produced, and the leaves of the proof tree are generalized and taken as the antecedent of a new rule for the target concept. Suppose, for instance, that the domain theory is the following

$$\begin{aligned} C(X) &:- B(X), A(X,Y). \\ B(X) &:- R(X,Y). \\ B(b). & \\ A(c,a). & \\ R(c,d). & \end{aligned}$$

and a positive example for C is c. It is then easy to see that $C(c)$ can be obtained deductively from the domain theory, where the leaves in the proof tree are $R(c,d)$ and $A(c,a)$. This produces the learned description for C:

$$C(X) :- R(X,Y), A(X,Z)$$

The learned description is actually a logical consequence of the given domain theory, and could be obtained from the domain theory even without looking at the examples, by resolving the first two clauses in the domain theory. The role of the example is that of suggesting some consequences of the domain theory rather than others: the ones that are useful to deduce the positive example are chosen.

One natural question arises: what is the purpose of this form of learning, if its output is just a logical consequence of something that was already known? The answer is that, although nothing really new is learned, the new form of knowledge may be more *operational*, that is to say either easier to use or leading to more efficient computations. In the above case, the positive example c, that is to say $C(c) =$ true may be deduced more efficiently from the learned clause than by the original domain theory, as one resolution step is saved. EBL may then be seen as a form of pre-compilation, or partial evaluation. The effectiveness of this form of learning speed-up depends on whether the given positive examples are representative of future cases: if this is not true, the learned clause may turn out to be useless, as the original domain theory would be necessary every time. Learned clauses that are useless in this sense will only take away some memory space without giving any computational advantage.

For this reason, systems soon started using EBL with many positive examples, so that only clauses that frequently proved to be useful would be kept. Other systems (e.g., [Bergadano and Giordana, 1988]) also introduced the possibility of using negative examples, in connection with the acceptance of a domain theory that might be partly incorrect. In this case, even clauses that follow deductively from the domain theory may be incorrect, and may cover negative examples. Clauses covering too many negative examples may then be discarded. When used in this way, the domain theory basically defines a hypothesis space. The concept descriptions that are logical consequences of the domain theory are descriptions that may potentially be learned. The description which is actually produced would also be required to perform well with respect to the data, i.e. cover many positive examples and few negative examples. EBL may then be considered as a way to guide search in top-down specialization: not all specializations are possible, but only the ones that produce deductive consequences of the domain theory. Legal specializations are obtained by resolving the clause to be specialized against some clause in the domain theory. Other specializations are not considered. In the previous example, the clause

$$C(X) :- R(X,Y), A(X,Z)$$

would be a legal specialization of the clause

$$C(X) :- B(X), A(X,Y).$$

while, e.g., the clause

$$C(X) :- B(X), A(X,Y), B(Y)$$

would not be considered. If the space of possible clauses is too large, a domain theory can be used effectively to prune such a space and suggest preferred specialization steps. ML-SMART [Bergadano and Giordana, 1988], and FOCL [Pazzani and Kibler, 1992] adopt similar goals and techniques.

Abduction comes into place if the consequences of the domain theory are not just taken to be deductive consequences, but are also obtained by means of an abductive theorem prover. For instance, if the above domain theory would contain the clause

$$D(X) :- A(X,Y)$$

then, an abductive/deductive consequence of the theory would be the clause

$$C(X) :- R(X,Y), D(X)$$

which would also be a legal specialization of the clause

$$C(X) \quad :- \quad B(X), A(X,Y).$$

Such an abductive use of a domain theory is found in [Bergadano *et al.*, 1989], with the purpose of restricting the hypothesis space without excluding concept descriptions that may be meaningful on the basis of abductive reasoning. In fact, such a use of abduction is just a means for specifying in a declarative way a form of *inductive bias*, a criterion for preferring an inductive hypothesis over another, or a rule for eliminating some description from the space of possible inductive generalizations. As such a criterion is heuristic in nature, one would not see why a deductive approach would be more justified than an abductive one. The deductive approach is the one found in ML-SMART and other approaches to the integration of inductive reasoning and EBL. The abductive approach follows exactly the same scheme, and is described, for example in [Bergadano *et al.*, 1989]. In this case the rule "accept generalizations that are a logical consequence of a given domain theory" is replaced by "accept generalizations that may be obtained from the domain theory via abductive reasoning". In both cases the obtained generalizations must *in primis* be consistent and complete with respect to the given examples, or satisfy such requirement *to a degree*. The notion of *abductive EBL* in a similar framework is introduced in [Cohen, 1992; Cohen, 1994].

4 RELATIONAL LEARNING ALGORITHMS BASED ON EXTENSIONALITY

Relational Learning Algorithms learn recursive relational descriptions from positive and negative examples, given as ground literals. Usually, a subset of the first-order predicate calculus, Horn clauses, is used for the concept description (see, e.g. [Birnbaum and Collins, 1991; Rouveirol, 1992; Cohen, 1992; Bergadano and Gunetti, 1993; Muggleton and Feng, 1990]). Learning definite clauses for multiple predicates is difficult, and systems tend to be slow. Many systems, such as Foil [Quinlan, 1990],

Focl [Pazzani and Kibler, 1992] and Golem [Muggleton and Feng, 1990] handle this problem by evaluating clauses extensionally. In this way candidate clauses can be generated directly from the examples one at a time and independently of one another.

Let us now introduce the concept of extensionally defined predicate, which, intuitively, will play the role of the basis case in the recursive definition of more complex predicates.

DEFINITION 1. A predicate $P(X)$ is said to be extensionally defined if it is defined over a given finite collection of elements $a_1, \ldots, a_n$, that is to say, if its characteristic set is given extensionally.

Examples of extensionally defined predicates are given by "mother" and "father" defined by the following table:

Parenthood Table

mother			father	
X	Y		X	Y
d	g		f	g
r	d		s	d
r	p		s	p
i	r		b	r
t	s		h	c
a	c			

Some other predicates can be intensionally defined over a collection of extensionally defined predicates by means of simple logical rules.

An example is given by the predicate *parent*:

$$\text{parent}(X,Y) \quad :- \quad \text{mother}(X,Y)$$
$$\text{parent}(X,Y) \quad :- \quad \text{father}(X,Y)$$

In this case, the extensional definition of the predicates *mother* and *father* provides as well an extensional definition of the predicate *parent*.

We formalize now the meaning of "extensional evaluation".

DEFINITION 2. Let $\alpha(X_1, X_2, \ldots, X_n)$ be a conjunction of predicates where there is at least one occurrence of each of the variables (not necessarily distinct) $X_1, \ldots, X_n$. We say that $\alpha(a_1, a_2, \ldots, a_n)$ is *extensionally true* if and only if

- in case $\alpha(X_1, \ldots, X_n) \equiv Q(X_1, \ldots, X_n)$, for some extensionally defined predicate Q, then $Q(a_1, \ldots, a_n)$ is a given positive example of Q;
- in case

$$\alpha \equiv \gamma(X_{i_1}, \ldots, X_{i_h}, T_1, \ldots, T_l), Q(T_1, \ldots, T_l, X_{j_1}, \ldots, X_{j_k}),$$

for some conjunction γ, predicate Q, $l \geq 0$ and

$$\{X_{i_1}, \ldots, X_{i_h}, X_{j_1}, \ldots, X_{j_k}\} = \{X_1, \ldots, X_n\},$$

then there must exist $e_1, \ldots, e_l$ such that both the conjunction $\gamma(a_{i_1}, \ldots, a_{i_h}, e_1, \ldots, e_l)$ and $Q(e_1, \ldots, e_l, a_{j_1}, \ldots, a_{j_k})$ must be extensionally true.

We say that the conjunct $\alpha(X_1, \ldots, X_n)$ *extensionally covers* a positive [resp. negative] example of P, $P(a_1, \ldots, a_n)$ [resp. $\neg P(a_1, \ldots, a_n)$] if and only if $\alpha(a_1, \ldots, a_n)$ is extensionally true.

The next subsection contains a simplified version of Foil, in order to illustrate the extensional approach and its drawbacks.

4.1 *The Extensional Approach: Simplified Foil*

Let P be the target concept and pos_examples(P) and neg_examples(P) the given positive and negative examples of P (in the following, α and γ represent generic conjunction of literals).

Algorithm 1 (Simplified Foil).

Let $E^+(P) =$ positive examples of P;

Let $E^-(P) =$ negative examples of P;

Let Π be the empty program;

while $E^+(P) \neq \emptyset$ **do**

Generate one clause $P \ :- \ \alpha$ and add it to Π;

Let $E^+(\alpha)$ be the positive examples of P covered by α;

$E^+(P) \leftarrow E^+(P) \setminus E^+(\alpha)$

end while

Output Π

Procedure **Generate one clause**

$\alpha \leftarrow$ true;

let $E^-(\alpha)$, $E^+(\alpha)$ be respectively the negative and positive examples of P covered by α;

while $E^+(\alpha) \neq \emptyset$ **do**

if $E^-(\alpha) = \emptyset$ then return($P \ :- \ \alpha$)

else choose a predicate Q and its arguments $\vec{X}$

$\alpha \leftarrow \alpha \wedge Q(\vec{X})$

end while

end Algorithm.

Every predicate Q can be defined by the user either intensionally by or extensionally. In particular, clauses can be recursive and, in this case, $Q = P$, and its truth value can only be determined by the available examples.

The choice of the literal $Q(\vec{X})$ to be added to the partial antecedent α of the clause being generated is guided by heuristic information. It might nevertheless be a wrong choice in some cases, in the sense that it may cause the procedure "Generate one clause" to fail by exiting the while loop without returning any clause. This problem can be fixed by making the choice of $Q(\vec{X})$ a backtracking point.

Suppose, for instance, that Foil is given the following positive and negative examples of the relation *ancestor*, which is the one to be learned:

$+ < r, g >, < b, g >, < b, d >, < d, g >, < b, r >, < r, d >, < r, p >;$

$- < d, d >, < g, p >, < d, p >.$

Let us suppose as well that we have the intensional definition for the relation *parent* given above along with the Parenthood Table which extensionally defines the predicates mother and father. Finally, we know that the logic program for *ancestor* depends on *parent* and on itself (i.e. it may be recursive). As there are at most 3 variables to be used, these are the possible literals:
parent(X, Y), parent(Y, X), parent(X, W),
parent(W, X), parent(Y, W), parent(W, Y)
ancestor(X, W), ancestor(W, X), ancestor(Y, W), ancestor(W, Y) [1].

The learning algorithm starts to generate the first clause - the antecedent α is initially equal to "true". We need to choose the first literal $Q(\vec{X})$ to be added to α. As we have left the heuristics unspecified, we will choose it so as to make the discussion short.

Let $\alpha =$ parent(W, Y); then all positive examples and the second and third negative examples are covered, so more literals need to be added.

Let $\alpha =$ parent(W, Y) $\wedge$ parent(W, X); in this case no positive examples are covered and the negative example $< d, p >$ is covered. Clause generation fails and we backtrack to the last literal choice.

Let $\alpha =$ parent(W, Y) $\wedge$ ancestor(X, W); no negative example is covered, and the first 3 positive examples are extensionally covered. A clause is generated and the covered positive examples are deleted.

We proceed to the generation of another clause; α is empty again. If we choose the first literal as parent(X, Y), the remaining positive examples are covered and the final solution is obtained:

ancestor(X, Y) $:-$ parent(W, Y), ancestor(X, W).
ancestor(X, Y) $:-$ parent(X, Y).

Notice that the obtained definition of the relation "ancestor" is complete and consistent when completeness and consistency are characterized as follows.

DEFINITION 3. A definition Π of P is *complete* with respect to the positive examples E^+ of P if and only if

$$(\forall e \in E^+)\Pi \vdash P(e).$$

A definition Π of P is *consistent* with respect to the negative examples E^- if and only if

$$(\forall e \in E^-)\Pi \nvdash P(e).$$

[1] ancestor(X, Y) and ancestor(Y, X) are not listed because they may produce looping recursions

4.2 *Problems of extensionality*

The independence of the clauses in the generation of the definition of "ancestor" is given by the extensional interpretation of recursion and sub-predicates: when a predicate Q occurs in a clause antecedent α, it is evaluated as true when the arguments match one of the positive examples. For instance, the clause

$$\text{ancestor}(X,Y) \ :- \ \text{parent}(W,Y), \text{ancestor}(X,W).$$

was found to extensionally cover the positive example $< b, g >$ of ancestor because parent(d, g) is true (see Parenthood Table) and $< b, d >$ is also a positive example of ancestor. The previously generated logical definitions of Q are not used. The method is (partially) justified by the following result (see [Bergadano and Gunetti, 1993])

THEOREM 4. *Suppose Foil successfully exits its main loop and outputs a logic program* Π, *that always terminates for the given examples.*

Let $Q(X) \ :- \ \alpha$ *be a generated clause of* Π, *then, for any* e *positive example of* Q, *if* α *extensionally covers* e *then* $\Pi \vdash Q(e)$.

However, extensionality forces us to include many examples, which would otherwise be unnecessary. In fact other desirable properties, similar to the one given by Theorem 4, are not true, and two fundamental problems arise:

PROBLEM 5. For a logic program Π, it may happen that $\Pi \vdash Q(e)$, but none of its clauses extensionally cover e. As a consequence Foil would be unable to generate Π.

Consider for instance the program Π:

$$\begin{aligned} \text{ancestor}(X,Y) \ &:- \ \text{parent}(W,Y), \text{ancestor}(X,W). \\ \text{ancestor}(X,Y) \ &:- \ \text{parent}(X,Y). \end{aligned}$$

Let $< r, g >$ be the only positive example of ancestor. As it is easy to see, by using resolution on Π and the Parenthood Table this example logically follows from Π ($\Pi \vdash$ ancestor(r, g)). However, is not extensionally covered. Indeed, by inspecting the Parenthood Table, we can see that the second clause does not cover it because parent(r, g) is false and the first clause does not cover it extensionally because parent(d, g) is true, but $< r, d >$ is not given as a positive example of ancestor.

PROBLEM 6. Foil may generate definitions which are not consistent. In details, even if no clause of a definition Π extensionally covers a negative example e of Q, it may still happen that $\Pi \vdash Q(e)$.

Consider for instance the following definition Π for "ancestor":

$$\begin{aligned} \text{ancestor}(X,Y) \ &:- \ \text{parent}(W,X), \text{ancestor}(W,Y) \\ \text{ancestor}(X,Y) \ &:- \ \text{parent}(X,Y). \end{aligned}$$

It follows that $\Pi \vdash$ ancestor(g, p). Nevertheless, $< g, p >$ is not extensionally covered by the first clause: parent(d, g) is true, but $< d, p >$ is not a positive example of ancestor.

Since d is not an ancestor of p, it could not possibly be added as a positive example, and $< g, p >$ would not be extensionally covered even if all positive examples were given.

The solution for this problem differs from the one of problem 1: in this case Π will be ruled out by adding a negative example, namely $< d, p >$.

5 PROBLEMS OF INTENSIONAL METHODS

Giving up the extensional interpretation of predicates while keeping the basic computational structure of Foil produces unsolvable problems. In fact, the truth values of the missing examples could be obtained by means of the partial program generated at a given moment. But if the inductive predicates occurring in a clause being generated are evaluated by means of the clauses that were learned previously, then the order in which the clauses are learned becomes a major issue.

Suppose, for instance, that we are given as family tree the following subset of the Parenthood Table:

$$\text{parent}(i,r), \text{parent}(r,d), \text{parent}(d,g), \text{parent}(f,g)$$

as well as the following positive and negative examples of ancestor:

$$\begin{aligned} + \ & < i,d >, < i,r >, < i,g >; \\ - \ & < i,f > . \end{aligned}$$

Suppose also that the following clauses are generated, with the given order:

(1) ancestor$(X, Y) \ :-$ parent(X, W), parent(W, Y).

(2) ancestor$(X, Y) \ :-$ parent(Y, W), ancestor(X, W).

(3) ancestor$(X, Y) \ :-$ parent(X, W), ancestor(W, Y).

Clause (1) does not use any inductive predicate and immediately covers the positive tuple $< i, d >$. Clause (2) contains ancestor(X, W), and this literal is evaluated with the clauses available at this stage, i.e. (1) and (2). With this kind of intensional interpretation, the only example covered is $< i, r >$, a positive example. When clause (3) is generated, it will cover the positive example $< i, g >$.

But clause (2) will now cover the negative example $< i, f >$, since parent(f, g) is true, and ancestor(i, g) may now be deduced by using the third clause.

Rules that seem consistent and useful at some stage may later be found to cover negative examples. We must then backtrack to the generation of the clause causing the inconsistency, e.g. clause (3), and to the generation of the clause that was later found to be inconsistent, e.g. clause (2). In general, we must give up our former assumption that clauses may be learned one at a time and independently. Alternatively, we may number the predicates occurring in the learned clauses every time an inconsistency is detected. For instance, the former program would be rewritten as

$$\begin{aligned}
\text{ancestor1}(X,Y) &:- \text{parent}(X,W), \text{parent}(W,Y).\\
\text{ancestor2}(X,Y) &:- \text{parent}(Y,W), \text{ancestor2}(X,W).\\
\text{ancestor3}(X,Y) &:- \text{parent}(X,W), \text{ancestor3}(W,Y).\\
\text{ancestor}(X,Y) &:- \text{ancestor2}(X,Y).\\
\text{ancestor2}(X,Y) &:- \text{ancestor1}(X,Y).\\
\text{ancestor}(X,Y) &:- \text{ancestor3}(X,Y).\\
\text{ancestor3}(X,Y) &:- \text{ancestor1}(X,Y).
\end{aligned}$$

Nevertheless, this technique does not totally avoid the need for backtracking, and explodes the number of possible clauses by multiplying the number of inductive predicates by the number of generated indexes. A solution based on abduction is presented in the next section, so that the advantages of extensionality are preserved, i.e. so that previously generated clauses do not need to be reconsidered.

6 COMPLETING EXAMPLES VIA ABDUCTION BEFORE LEARNING

There is no reason why particular positive (problem 1) or negative (problem 2) examples should have to be present. After all, the whole motivation of induction is that some information is missing. The important points are that

(1) if a definition Π consistent with the given examples exists, then it must be found; and

(2) the induced definition Π must not cover any negative examples.

The extensional approach guarantees neither, unless some specially determined positive and negative examples are given.

Here we show how abduction can be used to query the user for the missing examples, in order to preserve the computational advantages of extensional approaches, while guaranteeing that a correct solution be found.

Suppose that a learning system has to cover the positive example $P(a, b)$ and that the following candidate clause has been generated:

$$P(X,Y) \ :- \ \alpha(X,Y), Q(Y,Z).$$

Moreover, we know that $\alpha(a, b)$ is true (for example because every literal in α is extensionally evaluated to true). Then, that clause will extensionally cover $P(a, b)$ only if there exists some value c such that $Q(b, c)$ is a positive example (known to the system) of Q. Suppose that all such examples are missing. From the classical abductive inference rule:

$$\frac{\alpha \leftarrow \beta \qquad \alpha}{\beta}$$

we get:

$$\frac{P(a,b) \ :- \ \exists Z \alpha(a,b), Q(b,Z). \qquad P(a,b)}{\exists Z Q(b,Z)}$$

and then the user can be queried for a value of Z such that $Q(b, Z)$ is true. The new example of Q is added to the set of positive examples and $P(a, b)$ can now be covered. This is a *controlled* form of abduction, in the sense that the result of abductive inference is not asserted as true, but only proposed as a possible truth, which is then queried to the user. It is clear that if there is no value Z such that $Q(b, Z)$ is true then the clause must be discarded.

The above basic idea of generating abductive queries can be applied in a systematic way. That is, we may build chains of abductive queries, or chains of controlled abductive inference steps. A backward reasoning from an initial example that we know to be true (and that must be extensionally covered), up to what must be true in order for that initial example to be covered. The chain of abductive steps can be depicted as in Figure 1.

That is, a chain is produced because the queried examples are treated in the same way as the starting one. The added examples are used to generate other abductive queries again and again until no more examples can be added, and the backward reasoning can stop. We show this on a very simple example.

Let the following hypothesis space HS be given, containing only the correct clauses for the *append* concept (or program):

```
{ c1 = app(X,Y,Z) :- head(H,H),tail(X,T),app(T,Y,W),cons(H,W,Z).
  c2 = app(X,Y.Z) :- null(X),equal(Y,Z). }
```

Where all the predicates (except for *app*) are intensionally defined as usual. Let the only given positive example be: e_1 = "app([a,b],[c],[a,b,c])". A correct program for *append* cannot be extensionally learned from HS using only this example. However, let us apply the abductive queries as described above. Example e_1 is matched against the head of c1 producing the query "app([b],[c],W)?". That

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha \leftarrow \beta \qquad \alpha}{\beta}}{\Downarrow}}{\beta \leftarrow \gamma \qquad \beta}}{\gamma}$$

$$\Downarrow$$

$$\dfrac{\gamma \leftarrow \delta \qquad \gamma}{\delta}$$

$$\Downarrow$$

$$\vdots$$

Figure 1. A chain of abductive steps

is, the user is queried for a value of the variable W such that app([b],[c],W) is a positive example of *append*. Since the answer is "W = [b,c]", the new example e_2 = "app([b],[c],[b,c])" is added to the set of known examples of *append* and treated in the same way. It is matched against c1 producing the new example e_3 = "app([],[c],[c])". Now, e_3 is handled in the same way, but it cannot be extensionally covered by c1, as the evaluation of "head([],H)" fails. So, it is matched with the head of clause c2 that is found to cover the example. Since no more examples are added and all the examples are extensionally covered by some clause in HS, the abductive reasoning can stop and so also the learning task. Clauses c1 and c2 have been learned.

In fact, the effect of building a chain of controlled abductive steps is that of producing a derivation tree in reverse order for the starting example e. That is, from e to the leaves of the tree, that will be clauses from the background knowledge, known (or queried) examples, and clauses from the hypothesis space (i.e. learned clauses).

If the same is done for each positive example given initially, and recursively for each queried positive example, then the first problem of extensionality is solved. If there exists a derivation tree for a positive example in the set {hypothesis space + background knowledge + known examples} such a tree is found, and together with it also the clauses necessary to the derivation. The found clauses belonging to the hypothesis space will be part of the description of the target concept.

However, the second problem of extensionality still remains. How can we avoid a (known) negative example to be derived by a learned description? In fact, the same strategy can be used, but in a complementary way. For each negative example, we test the existence of a derivation tree for it. If such a tree is found, clauses from the hypothesis space involved in the derivation will not be part of any description of the target concept. Let us show this with an example.

Consider the following (wrong) program P for *reverse*:

c1 = reverse(X,Y) :- head(X,H), tail(X,T1), head(Y,H), tail(Y,T2), reverse(T1,T2).
c2 = reverse(X,Y) :- null(X), null(Y).
c3 = reverse([X,Y,Z],[Z,Y,X]).

that can be learned by using this set of examples:

e_1 = reverse+([],[]), e_2 = reverse+([1],[1]), e_3 = reverse+([3,2,1],[1,2,3]).
e_4 = reverse-([3,2,1],[3,2,1]).

Then P $\vdash$ reverse([3,2,1],[3,2,1]), however, P is extensionally consistent with the provided examples. We can avoid learning P by means of a chain of abductive queries as follows: e_4 is matched against the head of c1 generating the query "reverse([2,1],[2,1])?" that is classified as a negative example (let call it e_5) by the user. Then, e_5 is matched against c1 and the clause is found to extensionally cover this example, since "reverse([1],[1])" is a known positive example. As a consequence, c1 extensionally covers a negative example (a queried one, indeed) and is rejected.

On the ground of the above examples, we have devised the following strategy. Every legal, i.e. permitted by the constraints of the learning scheme, clause of type

$$P(X,Y) \ :- \ \alpha(X,Y,Z_1,\ldots,Z_k).$$

is processed with the following Abductive Completion Procedure:

ACP

for every example $P(a,b)$ of P do

(1) ask the user for a set of values $c_1,\ldots,c_k$ for $Z_1,\ldots,Z_k$ such that $\alpha(a,b,c_1,\ldots,c_k)$ is true;

(2) for every $Q(\cdot)$ occurring in α as a conjunct, add the obtained example to the collection of its positive examples.

end Algorithm.

Notice that adding one example may cause the request of others. As a consequence, the procedure must be repeated for every clause, over and over, until no more examples are added.

Consider, for instance, the example about the ancestor relation given above, and the following two clauses:

$$\text{relative}(X,Y) \ :- \ \text{ancestor}(X,Y).$$
$$\text{ancestor}(X,Y) \ :- \ \text{parent}(W,Y), \text{ancestor}(X,W).$$

where

$< b, g >$ is the only positive example of "relative",

$< d, d >$ is the only negative example of "relative", and

$< f, f >$ is the only negative example of "ancestor".

By using the first clause, the user is queried for ancestor(b, g), and this is added to the positive examples of "ancestor". Using the second clause, the user is asked for a value of W such that parent(W, g) and ancestor(b, W) are both true. By inspecting the Parenthood Table we have the answer $W = d$.

The second clause can be used again with $X = b$ and $Y = d$, and we obtain the completed set of positive examples for "ancestor":

$$+ < b, g >, < b, d >, < b, r >$$

Not all possible examples have been added, only the ones that were useful for those two clauses, given the initial examples. If this is done for all the clauses that are possible a priori, i.e. that satisfy the given constraints, then problems 1 and 2 no longer hold:

THEOREM 7. *Suppose the examples given to an extensional learning system are completed with the ACP. If the learning system successfully exits its main loop and outputs a definition* Π *(for a concept* P*,) then* $\Pi \vdash P(a, b)$ *implies that* $P(a, b)$ *is extensionally covered by some clause in* Π.

Proof. Suppose by contradiction that (c1) $\Pi \vdash P(a, b)$ but (c2) $P(a, b)$ is not extensionally covered.

Let $P(X, Y) \ :- \ \alpha(X, Y, Z1, \ldots, Z_k)$ be the clause resolved against $P(a, b)$ in a refutation of $: -P(a, b)$. Suppose that $P \vdash \beta(q, r) \wedge \gamma$ and $P \vdash R(r)$, but no such r is a positive example of R. There must be one literal $R(Y)$ having this property, because of assumptions (c1) and (c2). Since the tuples of R must have been completed with the given procedure, the user has been queried for $R(r)$, and this must have been inserted as a negative example. Therefore, it cannot be extensionally covered. We could now repeat the same argument for R. This would produce a non-terminating chain of resolution steps, and a finished proof of $Q(q)$ would never be obtained, contradicting the hypothesis that $P \vdash Q(q)$. ■

As an immediate corollary of the above theorem we have

COROLLARY 8. *If the examples given to an extensional learning system are completed with the ACP, and the learning system successfully exits its main loop and outputs a definition* P*, then* P *is consistent.*

It should be noted that this abductive completion is done as a preprocessing step. Then, it guarantees that a solution consistent with the examples is found if it exists. Moreover, it does not require reconsideration of previously generated clauses, as do systems (e.g. MIS [Shapiro, 1983]) that ask for new examples only when they are needed and during the learning process.

7 DEALING WITH FUZZY RELATIONS

It is quite interesting to notice that the relational learning framework described in the previous sections can, to some extent, be extended to the case where fuzzy relations are present. We recall that a fuzzy relation or a fuzzy predicate $P(\cdot)$ no longer has a binary value true or false, but can take on any value, denoted with $\tau(P(\cdot))$ from the interval $[0, 1]$. Historically, the semantic of vague (or fuzzy) predicates was introduced in [Zadeh, 1965]. Since it is way beyond our scope to provide here a complete formalization of fuzzy logic, for complete details refer to now classical texts such as [Dubois and Prade, 1980; Klir and Folger, 1988; Zimmerman, 1984] and to [Léa Sombé, 1990] for an interesting and in depth discussion of formalism for uncertain reasoning, including fuzzy logic.

In order to be able to deal with fuzzy predicates or relations we need to appropriately extend all the definitions introduced in the previous sections.

7.1 Fuzzy operators

In order to define a form of fuzzy abduction useful for our discussions we need to recall a few definitions.

- The standard logical operators can be extended in many ways, one possibility is the following

Fuzzy Logic Operators

$\neg P$	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$
$1 - \tau(P)$	$\max(\tau(P), \tau(Q))$	$\min(\tau(P), \tau(Q))$	$\min(\tau(Q) - \tau(P) + 1, 1)$

- The extension of Modus Ponens in fuzzy logic can be explained as follows

$$\tau(Q) \geq \min(\tau(P \rightarrow Q), \tau(P))$$

We can then propose the following fuzzy abductive inference rule

$$\frac{\alpha \leftarrow \beta \qquad \alpha}{\beta}$$

$$\tau(\beta) \leq \tau(\alpha)$$

7.2 *Learning fuzzy relations from examples*

It is clear that when we talk about learning fuzzy relations from examples, we are not going to consider positive or negative examples, but instead examples with a"degree of positivity", i.e. the label is a number between zero and one. As a subcase, we can also have binary relations whose definitions depend upon fuzzy predicates as well.

The concept of"extensional definition" of a predicate is not affected by its fuzzyness. As in the binary case, a predicate is extensionally defined if all its values are given. For instance, the unary predicate"old(x)" can be extensively defined over the universe where the parenthood table is defined. The meaning of extensional evaluation, though, needs some new formalization.

DEFINITION 9. Let $\alpha(X_1, X_2, \ldots, X_n)$ be a conjunction of (possibly fuzzy) predicates where there is at least one occurrence of each of the variables (not necessarily distinct) $X_1, \ldots, X_n$. Let $0 \leq \theta \leq 1$. We say that $\alpha(a_1, a_2, \ldots, a_n)$ is *extensionally θ-true* if and only if

- in case $\alpha(X_1, \ldots, X_n) \equiv Q(X_1, \ldots, X_n)$, for some extensionally defined predicate Q, then $Q(a_1, \ldots, a_n)$ is an example of Q with degree of positivity at least θ.
- in case

$$\alpha \equiv \gamma(X_{i_1}, \ldots, X_{i_h}, T_1, \ldots, T_l), Q(T_1, \ldots, T_l, X_{j_1}, \ldots, X_{j_k}),$$

for some conjunction γ, predicate Q, $l \geq 0$ and

$$\{X_{i_1}, \ldots, X_{i_h}, X_{j_1}, \ldots, X_{j_k}\} = \{X_1, \ldots, X_n\},$$

then there must exist $e_1, \ldots, e_l$ such that both the conjunction $\gamma(a_{i_1}, \ldots, a_{i_h}, e_1, \ldots, e_l)$ and $Q(e_1, \ldots, e_l, a_{j_1}, \ldots, a_{j_k})$ must be extensionally θ-true.

We say that the conjunct $\alpha(X_1, \ldots, X_n)$ *extensionally θ-covers* an example of P, $P(a_1, \ldots, a_n)$ if and only if $\alpha(a_1, \ldots, a_n)$ is extensionally θ-true.

Given the above definition and a collection of examples for P of type $< x, y >$, $\tau(P(x, y)) >$, we can act as follows:

- Sort in non-decreasing order $0 \leq \theta_1 \leq \cdots \leq \theta_n \leq 1$ the values of the examples;
- for each value $\theta_i > 0$ apply Algorithm 1 (Simplified Foil) by using the θ_i-covering definition. Basically θ_i is used as a threshold value. All examples with value at least θ_i will be considered as "positive" and the others as negative;
- if for some value θ_i the program terminates and outputs a logic program Π_i we check whether Π_i is also a"good" output for any θ_j with $j \neq i$.

- If so the final output will be Π_i otherwise there will be no output.

Basically, we are looking for a program Π such that for any example of type $(< x, y >, \theta)$, with $\theta > 0$ the following two conditions are satisfied

(C1) there exists a threshold value $\theta' \leq \theta$ for which $< x, y >$ is θ'-covered;

(C2) for any threshold value $\theta' > \theta$, $< x, y >$ is not θ'-covered.

Since, by fixing a threshold value, we reduce the fuzzy learning problem to the binary one we can extend definition 3 as follows

DEFINITION 10. Let $0 < \theta \leq 1$ and let $E^{+,\theta}$ and $E^{-,\theta}$ be respectively, the set of examples whose value is at least θ and the set of examples whose value is less than θ.

A definition Π of P is *θ-complete* with respect to the examples $E^{+,\theta}$ of P if and only if

$$(\forall e \in E^{+,\theta})\Pi \vdash P(e).$$

A definition Π of P is *θ-consistent* with respect to the examples $E^{-,\theta}$ if and only if

$$(\forall e \in E^{-,\theta})\Pi \nvdash P(e).$$

Therefore, we can formalize our learning goal by saying that we are looking for a definition Π of P which is θ-consistent and θ-complete for any θ in the range of values of the given examples.

For instance, suppose we want to learn the target concept TA Tall Ancestor. We use again the parenthood table as example. We also need to provide the truth values of the predicate tall(x) defined over the universe of the parenthood table. For instance suppose such values are

Tall Table

a	.3	b	.5
c	.6	d	.7
f	.8	g	.9
p	.1	r	.6
s	.6	t	.5

Suppose also that the set of examples is the following

> 0 $(< r, g >, .6), (< b, g >, .5), (< b, d >, .5), (< d, g >, .7), (< b, r >, .5), (< r, d >, .6), (< r, p >, .6);$

$= 0$ $(< d, d >, 0), (< g, p >, 0)(< d, p >, 0).$

Then by using $\theta_1 = .5$, Foil will output the following program

$$\begin{aligned} \text{TA}(X,Y) &:- \text{parent}(W,Y), \text{TA}(X,W). \\ \text{TA}(X,Y) &:- \text{parent}(X,Y), \text{tall}(X). \end{aligned}$$

It is easy to see that for any $\theta > 0$ such a program covers all the "positive" examples and no"negative" examples. However, differences in the given data may cause our new algorithm to fail producing an output. If for instance, we have as example $(< r, d >, .5)$ then the second clause will θ-cover it as a "positive" example for $\theta = .5$ but it will also cover it as a "negative" example for $\theta = .6$ since $\text{tall}(r) = .6$ and $\text{parent}(r, d)$ has value 1.

7.3 *Using the fuzzy abduction rule*

Obviously, the same problems that we described when dealing with binary relations are also present in the case of fuzzy relations. Moreover, we have some other problems related to the fuzzyness of some of the relations or predicates involved. Indeed, the user may have provided fuzzy truth values for such relations or predicates which are somehow inconsistent with the values attached to the examples. In this case, completing the set of examples using the proposed fuzzy abduction rule will also force the user to change if necessary the values which he/she has provided for the fuzzy predicates or relations. Indeed, suppose the system has to θ-cover the example $P(x, y) = \theta$ for some x, y and that the following candidate clause has been generated:

$$P(X,Y) \ :- \ \alpha(X,Y), Q(Y,Z).$$

Moreover, suppose that $\alpha(x, y)$ is extensionally θ'-true for every $\theta' \geq \theta$. Then, that clause

(case 1) will extensionally θ-cover $P(x, y)$ if there exists some value z such that $Q(y, z)$ is an example (known to the system) of Q with value θ; and

(case 2) it will not θ'-cover $P(x, y)$ for $\theta' > \theta$ only if there is no value z such that $Q(y, z)$ is an example (known to the system) of Q with value $\theta' > \theta$

Three cases are possible

(a) there exists one z such that $Q(z, y)$ has value θ;

(b) there exists w such that $Q(w, z)$ has value less than θ and greater than zero but there is no z such that $Q(z, y)$ has value θ;

(c) for any z $Q(z, y)$ has value greater than θ.

If (a) holds, using our fuzzy abduction rule we can ask for an example z for which $Q(z, y)$ has value θ and will add it to the list of examples. If (b) holds, the clause

can never θ-cover $P(x, y)$. Therefore, we would have a definition which is not θ-complete. However, the example might be covered at a lower threshold and for no threshold value a "negative" example will be covered. The user must decide at this point whether there is an inconsistency in the data given to the system or whether to discard the clause

If instead (c) holds, our fuzzy abduction rule will not return any value therefore the clause will be discarded.

8 CLASSIFICATION RULES WITH NON BINARY PREDICATES

Abductive completion of collection of examples during a learning process can be quite useful even when dealing with non-fuzzy classification rules that contain non binary predicates and more generally, fuzzy predicates or sets. A model for learning classification rules in such a setting has been proposed in [Bergadano and Cuttello, 1993; Bergadano and Cutello, 1995; Bergadano and Cutello, 1995]. The main goal of this model is to introduce a theoretical framework for tuning membership functions inside a non-fuzzy classification system so to make the classification system *probably, approximately correct* as defined in [Valiant, 1984]. We recall that the learning process in such a case requires that the learned classification system is approximately correct with high probability and that the induction procedure is polynomial.

Under the requirement that the membership functions of the fuzzy predicates to be learned occur in a structurally given classification system, some interesting results were obtained. Though the general learning problem is computationally hard, under meaningful restrictions either on the underlying probability distribution or on the syntactic structure of the classification systems, fuzzy membership functions can be quickly learned by observing positive and negative examples. The obtained membership functions are probably approximately correct within a particular classification problem. Therefore, with respect to such a classification problem future examples will be correctly classified with high probability.

8.1 *The learning model*

Let $\mathcal{C}$ be a given concept class over a universe $\mathcal{U}$. For the time being we will suppose that $\mathcal{U} \subseteq \Re^k$ for some integer k. Therefore, each element of $\mathcal{U}$ is a k-tuple of real numbers whose intended meaning is to characterize specific subparts of the system under study. Therefore, k is the global dimension of the problem.

Let $C \in \mathcal{C}$ and suppose a system S_m of classification rules for C is given in the following form:

$$\begin{aligned} C(x) &:- Q_1^1(p_{Q_1^1}(x)), \cdots, Q_{n_1}^1(p_{Q_{n_1}^1}(x)) \\ \cdots &\quad \cdots \quad \cdots \qquad\qquad \cdots \\ C(x) &:- Q_1^m(p_{Q_1^m}(x)), \cdots, , Q_{n_m}^m(p_{Q_{n_m}^m}(x)) \end{aligned}$$

where

- the predicates Q_i^j are taken from a set $\mathcal{P}$=$\{Q_1, ..., Q_n\}$ of given unary predicates;
- for all Q_j, p_{Q_j} is a projection function that returns the parameter which is of significance for Q_j.
- for every component i, the predicates $Q_1, ..., Q_{m_i}$ corresponding to i (i.e. such that p_{Q_j} returns the value of the i-th component for all $1 \leq j \leq m_i$) define a linguistic order.

Notice that

- it is possible that the same predicate occurs in different rules;
- concerning linguistic orders we have that the membership functions must be ordered in the following sense: if $j > i$ then for every $0 \leq \alpha \leq 1$ we have $\{x|Q_i(x) > \alpha\} \prec \{x|Q_j(x) > \alpha\}$, where $\prec$ is a suitably defined order relation.

The intuitive meaning of the rule system is that a given individual x is classified as a positive example for the concept C if one or more of the rule antecedents are *true* for x.

Therefore we can say that $x \in \mathcal{U}$ is a member of the concept C i.e. $C(x)$ is true if and only if

$$\tau\left(\bigvee_{j=1}^{m}\bigwedge_{i=1}^{n_j} Q_i^j(p_{Q_i^j}(x))\right) = 1$$

where τ is the the truth function. Since we are allowing some predicates in $\mathcal{P}$ to be fuzzy whereas C is not we make the following assumptions:

- A security parameter $0 < \theta < 1$ is given;
- $C(x)$ is θ-true if and only if

$$\tau\left(\bigvee_{j=1}^{m}\bigwedge_{i=1}^{n_j} Q_i^j(p_{Q_i^j}(x))\right) > \theta$$

and where the truth value above is computed according to the min-max semantic, i.e.

$$\tau\left(\bigvee_{j=1}^{m}\bigwedge_{i=1}^{n_j} Q_i^j(p_{Q_i^j}(x))\right) = \max_{j=1,...,m}(\min_{i=1,...,n_j}(Q_i^j(p_{Q_i^j}(x)))).$$

Given a set F=$\{\mu_1, ..., \mu_n\}$ of membership functions associated to the predicates in $\mathcal{P}$, the fuzzy classification system (FCS for short) (1) will be denoted by S_m^F. The notation S_m will then denote the collection of all possible FCS's S_m^F, which in turn can be characterized as the collection of all sets F of membership functions.

We will suppose that the unknown membership functions we are trying to learn are *convex*. In particular, the above implies that membership functions do not have local maxima. Moreover, concerning the linguistic order the convexity hypothesis implies that for every predicate Q the set $\{x|Q(x) > \alpha\}$ is an interval of the real line for every α. So, we can suppose that the order relation $\prec$ is the standard order relation on real intervals:

$$[a, b] \prec [c, d] \text{ iff } a \leq c \text{ and } b \leq d.$$

Such FCS's will be called *convex* fuzzy classification systems.

By definition, S_m is connected to a non fuzzy predicate, i.e. non fuzzy concept C. Our goal is to learn the membership functions from positive and negative examples of the concept C, in the hypothesis that a classification system S_m^F will be used with respect to the fixed threshold value θ. The obtained truth value will be denoted by $S_m^F(x)$. In the end, x is classified as a θ-positive example of C if $S_m^F(x) > \theta$.

8.2 *Learnability of FCS*

As proven in [Bergadano and Cuttello, 1993; Bergadano and Cutello, 1995], the problem of PAC-learning the class of convex FCS can be approached as follows:

- given ϵ and δ draw at least

$$\max(\frac{4}{\epsilon}\ln\frac{2}{\delta}, \frac{16m}{\epsilon}\ln\frac{13}{\epsilon}),$$

  examples from the distribution D;
- determine the hypercubes of $\Re^k$ where there are θ-positive examples only (the *positive hypercubes*); suppose the hypercubes are $m' \leq m$;
- use one rule of S_m to cover each θ-positive hypercube.
- project the m' positive hypercubes on the i-th component, obtaining $m_i' \leq m_i$ positive intervals $\{[a_1, b_1], ..., [a_{m_i}, b_{m_i}]\}$
- define Q_j^i as a trapezoidal fuzzy set *using* the above defined intervals.

The following result holds (see [Bergadano and Cuttello, 1993; Bergadano and Cutello, 1995]).

THEOREM 11. *The general learnability problem is $\mathcal{NP}$-hard.*

Similarly to any The theorem is proven by using the result proven in [Pitt and Valiant, 1988] that learning a K-term DNF by a K-term DNF is $\mathcal{NP}$-hard. We recall that a K-term DNF is a boolean formula of type:

$$\bigvee_{i=1}^{K} M_i$$

where the M_i are monomials, i.e. for all $i = 1, 2, \ldots, K$

$$M_i = x_{i_1} x_{i_2} \cdots x_{i_a} \cdot \bar{x}_{i_{a+1}} \cdots \bar{x}_{i_{n_i}}$$

In details the problem of learning K-term DNF by K-term DNF is reduced to the problem of learning S_K by S_K.

PAC learnability results are obtained by imposing some restrictions. In particular, if we assume that the underlying distribution is such that every rule *fires* at least once then the learning problem can be quite easily solved. Notice in particular that this assumption is quite reasonable, as we may suppose that the classification rule is given by an expert who will not include a rule in S_m if it is totally useless relatively to the *world* (i.e. the probability distribution of events) he/she lives in.

8.3 *Completing the set of examples*

All the results above cited were obtained under the following assumptions:

(a1) The teacher (i.e. the experts who provides the structure of the classification system and the labels for the examples) does not need to justify the rules in the system;

(a2) the classification system may not be minimal, i.e. there may be"redundant rules";

(a3) for each component the linguistic order is given.

In order to bypass the hardness result it is obvious that more power must be given to the learner. In particular, in [Bergadano and Cutello, 1995] it was introduced the notion of"equivalence query". The learner was allowed to guess and ask for counter examples. Using this powerful tool and still under assumptions (a1)-(a3), PAC-learnability was achieved.

We now propose a way to extend the results obtained and get rid of conditions (a1)-(a3). In details, the abductive completion procedure (ACP, section 6) in such a framework can be adjusted as follows:

for each rule

$$Q_1(p_{Q_1}(x)) \wedge \cdots \wedge Q_n(p_{Q_n}(x)) \rightarrow C(x)$$

occurring in the system, we ask for a positive example x such that

$$\max_{j=1,\ldots,m} (\min_{i=1,\ldots,n_j} (Q_i^j(p_{Q_i^j}(x))) = \min_{i=1,\ldots,n} (Q_i(p_{Q_i}(x)).$$

If no such a tuple x exists then the rule is dropped from the classification system. It is then clear that

- by forcing the teacher to justify the given rules, the system obtained is"minimized", that is to say rules which have no influence in the classification process (no matter what the value of θ is) are eliminated.
- using a polynomially bounded number of examples, we are able in polynomial time to produce a probably approximately correct classification system, i.e. we achieve PAC-learnability;
- finally, the learner no longer needs to be given the linguistic orders for each component.

9 AUTOMATIC ABDUCTION OF KNOWLEDGE: TESTING

We have seen as abduction can be used in conjunction with induction, in order to alleviate the intrinsic weaknesses of this learning paradigm. However, one may reasonable wonder at which conditions such approach is practical. There are two main issues to consider. First, the abductive procedure terminates only if there is a finite number of answers to each question. This is the case if we are in a finite domain, as in the case of the *ancestor* relationship. In the case of infinite domains, the approach is still acceptable if we work with functional relations. By imposing adequate input/output constraints [Bergadano and Gunetti, 1993], it is possible to work only with positive examples. For each input tuple of the target relation, there is only one positive example matching the tuple, and infinite negative examples. the system can be instructed to work only with the positive examples, assuming the negative examples being all the examples with the same input as the positive ones and different output values. As a consequence, for each query, the user must provide exactly one positive example, and no more.

Second, the number of queries to the user must not only be finite, but also quite small, unless a database of examples is available, and the queries can be handled automatically. This depends mainly on the size of the hypothesis space, i.e. on the number of clauses that can be used to generate queries. In fact, it is becoming more and more clear as learning complex first order theories is quite difficult, and a lot of knowledge must be used. As a consequence, the user must not be a passive agent, just providing the examples of the target relation. She must also be able to design a restricted hypothesis space on the basis of the sought relation [Bergadano and Gunetti, 1996; Van Laer *et al.*, 1994]. Also the number of examples provided initially may influence the number of queries. Experiments based on the presented

approach have shown as it is possible to learn complex relations by using very few representative examples of the target concept, and letting the abductive procedure asking for the missing ones. For example, it was possible to learn a program for inserting a key into a balanced tree (rebalancing the tree if necessary) by using an extensional approach and just one well chosen initial positive example. With a hypothesis space of 2^{24} clauses the user was queried for 15 more examples, and the program could be learned in 1831 seconds on a Sun Sparcstation 5. The program was composed of 9 clauses, eight of them recursive [Bergadano and Gunetti, 1996]. In general, it can be shown that an initial positive example is sufficient to learn all the clauses necessary to derive it, if the abductive completion is applied.

In this section we illustrate an interesting application of this combination of abduction and induction, outside the field of Machine Learning. We present a technique to test the correctness of a given program where, moreover, (some of) the test cases are automatically generated by the Abductive Completion Procedure through queries made to the program being tested.

The learning procedure described in Algorithm 1 outputs a set of Horn clauses. As well known, this can be regarded as a logic program. As a consequence, in the rest of this section we will use the term *Inductive Program Learning* (IPL) in place of concept learning, and we will speak of *Logic Programs* instead of concepts described through a set of Horn clauses.

9.1 Induction and Testing

Testing is the field of Computer Science concerned with the problem of detecting the presence of errors in programs by executing them on specific and well-chosen input data. A set of test data allowing us to discover all the errors in the program is called *adequate*. Though it can be formally proved that no algorithm exists to generate an adequate set of test data for an arbitrary program, testing is necessary and widely used in software development. It is practical and (relatively) easy to apply, and can give some (even very precise) information about the correctness of a program.

Testing can be seen as a way of distinguishing a program from all the possible (syntactically correct) alternatives to that program. This is very similar to IPL, where a program in the hypothesis space must be identified from among all the other possible programs on the basis of the given examples. More precisely, testing and IPL are, in some way, symmetric. The latter goes from examples to programs, whereas the former goes from programs to examples (input values).

Most of the time, a program is tested on the basis of the "correct program the user has in mind". It is common in this case to assume that although the correct program is unknown, there is a known *set of programs* that can be seen as alternative implementations and should contain at least one correct solution. This concept of program correctness leads to a practical definition of meaningful test data ([DeMillo and Offutt, 1991]):

DEFINITION 12. ([Howden, 1976]) A test set T for a program P is *reliable* (with respect to a correct program P_c) iff $[\forall x \in T \;\; P(x) = P_c(x)] \rightarrow P \equiv P_c$
If P contains errors, this will be shown by the test cases in T.

DEFINITION 13. Let there be given a test set T for a program P and a set $\mathcal{P}$ of alternative programs. We say that T is *adequate* with respect to $\mathcal{P}$ if and only if it is reliable with respect to every program in $\mathcal{P}$.

As a consequence, if $\mathcal{P}$ contains at least one correct program P_c, then the test set will also be reliable for P. Many different testing techniques are based on a relation between the program P to be tested and a set of possible alternatives P_c. In particular, fault-based methods [DeMillo and Offutt, 1991; King and Offutt, 1991] use a set of typical programming errors defined a priori, and alternative programs are obtained by inserting some of these errors (that is, possible program mutations) in P.

Testing based on the inductive learning of logic programs also relies on definition 13 of adequateness of a test set. As noted above, an intuitive symmetry between induction and testing can almost immediately be noticed [Weyuker, 1983]: induction is the inference from examples to programs; testing is the inference from programs to input values. Given a test set T of input values for a program P, the examples of P for T are defined as E(P,T)={<i,o> | i ∈ T and P(i)=o}. This notion is formalized by Weyuker as follows [Weyuker, 1983]:

DEFINITION 14. A test set T is *program adequate* for a program P if and only if P_I is inductively inferred from E(P,T) and $P_I \equiv$ P.

The intuitive meaning of the above definition is as follows: If, given a set E(P,T) of input/output examples, we inductively infer a program $P_I \equiv P$, then T is likely to be useful for testing the program P.

In the previous sections we have described a a basic learning method which is terminating and explicitly make use of a finite set of legal programs. More precisely, the hypothesis space is made of Horn clauses, and every possible subset of such Horn clauses is a program that could be learned by the induction procedure, on the basis of the given examples. We will call $\mathcal{P}$ the set of all possible programs. If there are n Horn clauses, there are 2^n possible legal programs in $\mathcal{P}$.

The learned program P_I must belong to $\mathcal{P}$. This is consistent with the fact that all the test case generation procedures that are not specification based refer (sometimes implicitly) to a set $\mathcal{P}$ of alternative programs. As a consequence, definition 14 can then be stated in terms of "sound and complete[2] IPL": T is adequate for P if and only if P is the only program that can be learned from E(T,P) with a sound and complete IPL method. The restriction to a finite set of alternative programs (hypothesis space) $\mathcal{P}$ seems to be acceptable for testing; for instance, all approaches

[2]Formally, a system is defined to be *sound* if it only outputs programs that are consistent w.r.t. the given examples. It is *complete* if it is able to find a consistent program whenever such a program exists in the given hypothesis space. As an instance, the learning procedure of Algorithm 1 (Simplified Foil), is neither sound nor complete, by virtue of Problems 1 and 2.

to fault-based testing such as mutation analysis, are based on this assumption.

The intuitive symmetry between induction and testing can now be made clear: induction is an inference from the pair <E(T,P), $\mathcal{P}$> to a program P in $\mathcal{P}$, whereas testing is an inference from <P, $\mathcal{P}$> to the test set T. In the next subsection this symmetry will be used in actually *generating* the adequate test cases for a (not necessarily logic) program.

9.2 *Abductive Test Case Generation*

Let us call *SFOIL* (Simplified Foil) the learning procedure of Algorithm 1. Let us call $SFOIL_{ACP}$ the *SFOIL* algorithm augmented with the Abductive Completion Procedure and with input/output constraints, as discussed at the beginning of this section. A sequence of inductions of programs from examples is used to generate test cases. Initially, there are no examples and, in the end, the generated set of examples will be adequate in the sense of definition 13.

Let P be the program to be tested. At any given moment, the examples generated so far are used to induce a program P′. New examples that distinguish P from P′ are added, and the process is repeated until no program P′ that is not equivalent to P can be generated. This procedure is described in more detail below.

Test case generation procedure:

Input: a program P to be tested,
a finite set $\mathcal{P}$ of alternative programs
Output: an adequate test set T

T $\leftarrow \emptyset$;
loop:
<P′,T> $\leftarrow$ $SFOIL_{ACP}$(E(T,P),$\mathcal{P}$)
if P′ = "fail" **then return** T
if ($\exists$ i) P′(i) $\neq$ P(i)
then T $\leftarrow$ T $\cup$ {i}
else $\mathcal{P} \leftarrow \mathcal{P}$ - P′
goto loop

In order for the above procedure to work, the learning algorithm $SFOIL_{ACP}$ must be sound and complete, as it is proved by theorem 7 and corollary 8. In the test case generation procedure, the test set T is initially empty. The main step in the loop consists of using $SFOIL_{ACP}$ to learn a program P′ that is consistent with the examples generated so far. P′ is then ruled out either by (1) adding an input value to T or by (2) removing it from $\mathcal{P}$. As a consequence, P′ will not be learned again. When $SFOIL_{ACP}$ cannot find a program P′$\in \mathcal{P}$ that is consistent with the examples, then the only programs with this property are P and those equivalent to it, i.e., the test set T is adequate. This is proved by the following:

THEOREM 15. *Let equivalence be decidable for programs in $\mathcal{P}$. Then the above test case generation procedure outputs an adequate test set T for P.*

Proof. Since equivalence is decidable for programs in $\mathcal{P}$, the procedure terminates, as the condition ($\exists$ i) P′(i) $\neq$ P(i) always produces an answer. Suppose, by contradiction, that the obtained test set T is not adequate. This means that there is P′ $\in$ $\mathcal{P}$ and an input i $\notin$ T such that P′(i) $\neq$ P(i), but ($\forall$ v) $\in$ T, P′(v) = P(v). However, just before termination, $SFOIL_{ACP}$ failed to induce a program P_I. Moreover, P′ is complete and consistent with respect to E(T,P), because ($\forall$ v) $\in$ T, P′(v) = P(v). Then, just before termination, and because $SFOIL_{ACP}$ is sound and complete, it must be the case that P′ $\notin$ $\mathcal{P}$. As a consequence, P′ must have been removed from $\mathcal{P}$ at some previous iteration. But, in that case, there must have been an input value i such that P′(i) $\neq$ P(i), and this value would have been added to T. This contradicts the assumption that i $\notin$ T and T is not adequate. ■

If a decision procedure is available for finding an input i such that P(i) $\neq$ P′(i), this is used directly in the above test generation method. As this is not true in general, examples are found by enumerating (in some random or ad hoc order) the possible inputs i, and stopping when an i is found such that P(i)=o$\neq$P′(i)=o′. This enumeration could also be guided by other test case generation techniques, such as path or functional testing. The requirement of decidable equivalence is of course not easily verified or accepted. is a major theoretical and practical issue in program testing. In the implementation of our test case generation method, we approximate it by means of its time-bounded semi-decision procedure. Except for this approximation, the system produces adequate test sets with respect to any finite class of programs $\mathcal{P}$. The reader may find all the details of this testing technique in [Bergadano and Gunetti, 1996b].

9.3 *An example: testing a* merge *program*

Consider the problem of merging two ordered lists, where we allow them to contain repeated elements. The output list must be ordered and should contain every element of the input lists only once. This problem requires a procedure for removing the elements that are repeated. Suppose we are given the following:

```
remove(X,[X|Y],Z) :- !, remove(X,Y,Z).
remove(_,Y,Y).
```

This does just as much as necessary: it removes the initial occurrences of an element in a list, e.g., remove(a,[a,a,b],[b]), but remove(a,[b,a],[b,a]). Let the program to be tested be the following:

P:

```
1) merge(X,Y,Z) :- null(X), Y=Z.
```

```
2) merge(X,Y,Z) :- null(Y), X=Z.
3) merge(X,Y,Z) :- head(X,X1), tail(X,X2), head(Y,X1), tail(Y,Y2),
                   merge(X2,Y2,W), remove(X1,W,T), cons(X1,T,Z).
4) merge(X,Y,Z) :- head(X,X1), tail(X,X2), head(Y,Y1), tail(Y,Y2),
                   X1<Y1, merge(X2,Y,W), remove(X1,W,T), cons(X1,T,Z).
5) merge(X,Y,Z) :- head(X,X1), tail(X,X2), head(Y,Y1), tail(Y,Y2),
                   X1>Y1, merge(Y2,X,W), remove(Y1,W,T), cons(Y1,T,Z).
```

In this program the first two clauses are wrong. In fact, any repeated element in the nonempty list will be present in the output. Let $\mathcal{P}$ be defined as any program made of any clause that can be built by using as head *merge(X,Y,Z)* and whose body is made of a subset of the literals occurring in P. As a consequence, $\mathcal{P}$ contains $2^{2^{19}}$ alternative programs, i.e., all possible subsets of the space of clauses. Among the subsets there are versions of the correct implementation of *merge*. A concise representation of $\mathcal{P}$ will be given as input to $SFOIL_{ACP}$. Of course, $\mathcal{P}$ is not generated explicitly. See [Bergadano and Gunetti, 1996b] for the implementative details.

The test case generation procedure starts with an empty test set T_0 of input values, and calls $SFOIL_{ACP}$. As E(T_0,P) contains no examples, the empty program P_0 is an acceptable output of $SFOIL_{ACP}$(E(T_0,P),$\mathcal{P}$).

Pairs of lists X and Y are then enumerated, so that
$P_0 \vdash \text{merge}(X, Y, Z')$, $P \vdash \text{merge}(X, Y, Z)$ and $Z \neq Z'$
The first such pair that is found is <X,Y>=<[],[]>; for this input, P_0 produces no output and P outputs Z=[]. The new test set is then $T_1 = \{<[],[]>\}$.

$SFOIL_{ACP}$(E(T_1,P),$\mathcal{P}$) is called again, yielding P_1:
merge(X,Y,Z) :- Y=Z.
This program is an acceptable output of $SFOIL_{ACP}$ because merge([],[],[]) is derived from it, and the output is the same as that of P.

Pairs of lists X and Y are enumerated, so that
$P_1 \vdash \text{merge}(X, Y, Z')$, $P \vdash \text{merge}(X, Y, Z)$ and $Z \neq Z'$. The first such pair that is found is:
<X,Y>=<[1],[]>; for this input, P_1 outputs Z=[] while P outputs Z=[1]. The new test set is then $T_2 = T_1 \cup \{<[1],[]>\}$.
$SFOIL_{ACP}$(E(T_2,P),$\mathcal{P}$) is called again, yielding P_2:
merge(X,Y,Z) :- X=Z.
This program is an acceptable output of $SFOIL_{ACP}$ because merge([],[],[]) and merge([1],[],[1]) are derived from it.

The test case generation goes on in this way generating new test cases and learning new programs, until no new program can be discovered and $SFOIL_{ACP}$ fails,

ending the test generation process. The complete set of learned test cases, with the corresponding outputs, is the following:

1: merge([],[],[])
2: merge([1],[],[1])
3: merge([2],[1],[1,2])
4: merge([2],[],[2]) – abduced by $SFOIL_{ACP}$ from P
5: merge([],[1],[1]) – abduced by $SFOIL_{ACP}$ from P
6: merge([1],[1],[1])
7: merge([1,1,2],[1],[1,2])
8: merge([1,1,2],[],[1,1,2]) – abduced by $SFOIL_{ACP}$ from P, it shows the error
9: merge([1],[2],[1,2])

Only nine examples have been required to locate the error, whereas many more would have been necessary in random testing, if there are many possible element values with respect to the average list length.

We see that some of the examples (one of them showing the error) are added as a consequence of the ACP. Consider, e.g., example 7. When $SFOIL_{ACP}$ is learning a program P_5 consistent with examples 1-7, it has to find a clause covering example 7. During its construction, it comes to the (partial) clause:

merge(X,Y,Z) :- head(X,X1), tail(X,X2), head(Y,X1), tail(Y,Y2), merge(X,Y2,W)

There is no example discovered so far covering literal merge([1,1,2],[],W) and, as consequence, the ACP queries the program P (that is, it runs it) for a value o such that merge([1,1,2],[],o) is true, getting the answer o=[1,1,2] and producing the new testing case represented by example 8 above.

It should also be noted that P does not belong to the hypothesis space $\mathcal{P}$; actually, this is not required by the test case generation system. In fact, the program to be tested could be used as a black box and could, in principle, be written in any programming language.

10 CONCLUSION

The relation between induction and abduction is briefly discussed in this paper. Our main goal was then to show the specific uses of abductive reasoning in Machine Learning. On one hand, it has been used to guide search in a top-down specialization framework related to Explanation-based Learning. On the other hand, we have also shown how abduction can be used to query the user for examples that may be missing. This means that the user must not provide all the needed examples to learn one definition at a time. He/she can forget some examples, and the abductive procedure will ask for them. Observe that only the examples really needed are queried, so it will not waste time trying to cover

useless examples. In many extensional systems [Muggleton and Feng, 1990; Quinlan, 1990] the user must provide all the examples at one time, and usually a superset of the examples needed is given, resulting in a lot of time wasted in covering useless examples. We have also highlighted a possible extension of such a learning framework to the case where fuzzy predicates occur. In such a case, the user not only must provide all the missing example but must also provide consistent fuzzy values for all the predicates involved.

ACKNOWLEDGEMENTS

The authors thank esprit project DRUMS II which supported part of this research.

F. Bergadano and D. Gunetti
Computer Science Department, University of Torino, Italy.

V. Cutello,
Mathematics Deptartment, University of Catania, Italy.

REFERENCES

[Bergadano and Besnark, 1994] F. Bergadano and P. Besnard. Abduction and induction by non-monotonic logics, *Workshop on Mathematical and Statistical Methods in Artificial Intelligence*, Udine, Italy, 1994.

[Bergadano and Cuttello, 1993] F. Bergadano and V. Cutello. Learning membership functions. *Proceedings of ECSQARU'93*, 2nd European Conference on Symbolic and Quantitative Approaches to reasoning and uncertainty, Granada, Spain, 1993. *Lecture Notes in Computer Science, Springer-Verlag*, Vol. 747, pp. 25–32, 1993.

[Bergadano and Cutello, 1995] F. Bergadano and V. Cutello. Learning fuzzy sets. In *Proceedings of EUFIT'95*, G. J. Zimmermann, ed. Aachen, Germany, 1995.

[Bergadano and Cutello, 1995] F. Bergadano and V. Cutello. Probably approximate correct (PAC) learning in fuzzy classification systems. *IEEE Transactions on Fuzzy Systems*, **3**, 473–478, 1995.

[Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. *Proceedings of the Fifth International Conference on Machine Learning*. pp. 305–317, Ann Arbor, MI, 1988.

[Bergadano *et al.*, 1989] F. Bergadano, A. Giordana and S. Ponsero. Deduction in Top-down Inductive Learning, *Proceedings of the Sixth International Conference on Machine Learning*. pp. 23–25, Ithaca, NY, 1989.

[Bergadano and Gunetti, 1993] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Chambery, France, 1993. Morgan Kaufmann.

[Bergadano and Gunetti, 1996] F. Bergadano and D. Gunetti. *Inductive Logic Programming: from Machine Learning to Software Engineering*. MIT Press, Cambridge, MA, 1996.

[Bergadano and Gunetti, 1996b] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology (ACM TOSEM)*, **5**, 1996.

[Birnbaum and Collins, 1991] L. Birnbaum and G. Collins, eds. *Proceedings of the International ML Conference, part VI: Learning Relations*. Morgan Kaufmann, 1991.

[Cohen, 1992] W. Cohen. Abductive explanation-based learning: a solution to the multiple inconsistent explanation problem. *Machine Learning*, **8**, 167–219, 1992.

[Cohen, 1994] W. Cohen. Incremental abductive explanation-based learning. *Machine Learning*, **15**, 5–24, 1994.

[Console and Saitta, 1993] L. Console and L. Saitta. Generalization in learning and abduction. technical report, University of Torino, 1994.

[DeMillo and Offutt, 1991] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. on Software Engineering*, **17**, 900–910, 1991.

[DeRaedt and Bruynooghe, 1991] L. DeRaedt and M. Bruynooghe. CLINT: A Multistrategy Interactive Concept-Learner and Theory Revision System. In R. S. Michalski and G. Tecuci, eds. *Proceedings of the Workshop on Multistrategy Learning*, pp. 175–190, Harpers Ferry, VA, 1991.

[Dubois and Prade, 1980] D. Dubois and H. Prade *Fuzzy Sets and Systems: Theory and applications*. Academic Press, NY, 1980.

[Ellman, 1989] T. Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, **21**, pp. 163–222, 1989.

[Flach, 1992] P. Flach. Simply Logical, John Wiley and Sons, 1992.

[Howden, 1976] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. on Software Engineering*, **2**, 208–215, 1976.

[Klir and Folger, 1988] G. J. Klir and T. A. Folger. *Fuzzy Sets uncertainty and Information*. Prentice Hall, Englewood Cliffs, NJ, 1988

[King and Offutt, 1991] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, **21**, 685–718, 1991.

[Léa Sombé, 1990] Léa Sombé *Reasoning Under Incomplete Information in Artificial Intelligence*. John Wiley and Sons Inc., 1990.

[Van Laer *et al.*, 1994] W. Van Laer, L. Dehaspe and L. DeRaedt. Applications of a logical discovery engine. Leuven, Belgium, 1994. Technical Report, Dept. of CS, Univ. of Leuven.

[Mitchell *et al.*, 1986] T. Mitchell, R. M. Keller and S. Kedar-Cabelli. Explanation-based generalization: a unifying view, *Machine Learning*, 47–80, 1986.

[Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990.

[Pazzani *et al.*, 1991] M. Pazzani, C. A. Brunk and G. Silverstein. A Knowledge-intensive approach to learning relational concepts. *Proceedings of the 8th International Conference on Machine Learning*, 1991.

[Pazzani and Kibler, 1992] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, **9**, 57–94, 1992.

[Pitt and Valiant, 1988] L. Pitt and L. G. Valiant. Computational limitations on learning from examples. *Journal of ACM*, **35**, 965–984, 1988.

[Quinlan, 1990] R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, **5**, 239–266, 1990.

[Rouveirol, 1992] C. Rouveirol, ed. *Proceedings of the ECAI Workshop on Logical Approaches to Learning*. ECCAI, Vienna, Austria, 1992.

[Shapiro, 1983] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[Valiant, 1984] L. Valiant. A theory of the learnable. *Communications of the ACM*, **27**, 1134–1142, 1984.

[Weyuker, 1983] E. J. Weyuker. Assessing Test Data Adequacy Through Program Inference. *ACM Trans. on Programming Languages and Systems*, **5**, 641–655, 1983.

[Zadeh, 1965] L. A. Zadeh. Fuzzy sets. *Information and Control*, **8**, 338–353, 1965

[Zimmerman, 1984] H. J. Zimmermann *Fuzzy Set Theory and its applications*. Kluwer-Nijhoff, Boston, 1984.